ChatGPT

Security & Compliance Review and Recommendations

TLS Encryption

Ensuring all client-server communications are protected by TLS is critical. The application should **enforce HTTPS for all traffic**: - **Force HTTPS:** Configure the Flask app or the hosting environment to redirect any HTTP requests to HTTPS. This prevents sending sensitive data (PHI or credentials) over unencrypted connections. Enabling HSTS (Strict-Transport-Security header) is recommended so browsers only use HTTPS. - **Secure Cookies:** Continue using secure cookies – the app's config already marks session cookies as secure and HTTP-only [1] (ensuring they're only sent over HTTPS and not accessible to JavaScript). In development this is disabled [2], but in production it must remain enabled. - **Verify TLS on External Connections:** When the app calls external services (e.g. Epic's FHIR API), ensure those HTTP clients verify SSL certificates. Use `https://` endpoints for all external URLs.

By enforcing TLS in transit, you greatly reduce the risk of eavesdropping or man-in-the-middle attacks on sensitive health data.

HIPAA Compliance and BAA

Because this system handles PHI, it **must meet HIPAA requirements**. A major concern is the hosting platform: - **Use HIPAA-Compliant Hosting:** Replit is **not** HIPAA compliant and will not sign a Business Associate Agreement [3]. This is a blocker for using any PHI on Replit. To comply, consider migrating to a platform that provides HIPAA-compliant infrastructure and will sign a BAA (e.g. AWS, GCP, Azure, or specialized HIPAA hosting). If staying on Replit for development, **do not use real PHI** – only use dummy data until a compliant host is in place. - **Encryption at Rest:** Enable encryption for data at rest. If using an external database or disk storage, use encrypted volumes or built-in DB encryption. HIPAA technically permits unencrypted DBs if access controls are strict, but encryption at rest provides safe harbor in case of physical breaches. For example, if using PostgreSQL on a cloud host, enable encryption or use a managed service that encrypts storage. - **Audit Logging:** Continue the comprehensive audit logging already implemented. The system logs admin actions with user, org, IP, and details [4], which is excellent for HIPAA. Ensure these logs are **immutable** (administrators shouldn't be able to tamper with them) and retained for at least 6 years (HIPAA's recommendation for audit log retention, your code uses ~7 years by default [5]). - **Access Controls (PHI Minimization):** Restrict access to PHI to only those roles that need it. It appears the app already isolates data per organization and by user roles. Continue to enforce the principle of least privilege so, for example, an MA or nurse can only view their patients' data, and cannot access other organizations' data. - **No PHI in Logs or Emails:** Review all logging and debugging output to ensure no sensitive personal health info is inadvertently logged. The app has a PHI filter for documents and a testing route for it, which is good. Similarly, if the system sends emails (e.g. user invites), do not include PHI in email content and use a HIPAA-compliant email service or encryption if needed.

Ultimately, achieving HIPAA compliance will likely require moving off Replit (or any non-BAA environment) [3]. All third-party services that handle PHI (databases, email/SMS services, error monitoring like Sentry, etc.) should provide BAAs. The above steps, combined with the app's existing audit controls and role-based protections, put you on the right track for HIPAA compliance.

Database Storage & Multi-Tenancy

The application employs a multi-tenant data model, which is important for isolating data between organizations. Some improvements and confirmations for **secure database storage**: - **Use a Robust DB Backend:** Make sure the deployment is using PostgreSQL (which is listed in the Replit configuration) rather than SQLite. SQLite is used as a fallback [6], but for multi-user production PostgreSQL is needed for reliability and concurrency. Confirm that the `DATABASE_URL` is properly set so that `SQLALCHEMY_DATABASE_URI` isn't falling back to the file-based SQLite. - **Enforce Multi-Tenancy in Queries:** The code defines an `OrganizationScope` utility to filter queries by `org_id`, and models like `User` and `Patient` have unique constraints per org [7]. This is good. Continue to use these patterns for **every** data access. For instance, whenever querying patients, documents, etc., ensure you filter by the current user's `org_id` (the app does this in many places). The decorators like `@require_organization_access` help prevent cross-org access at the route level [8] – keep using those on any new routes. - **Data Partitioning:** For further isolation, consider additional safeguards like row-level security policies in PostgreSQL or even separate schema or databases per tenant if the scale or security needs increase. The current single-db with org_id approach is acceptable given the app logic prevents cross-access. - **Data Retention and Deletion:** Have a strategy for removing or archiving data that is no longer needed. HIPAA minimum is 6-year retention for medical records access logs, but you might not need to retain old patient records indefinitely if not required. Ensure when organizations are deleted, their data (patients, docs) is either deleted or anonymized as appropriate. The `delete_organization` route checks for no users [9] before deletion, but you might also need to delete related PHI like patients and documents for that org (and ensure backups are handled). - **Backups:** Implement regular backups of the database and **securely store** those backups (encrypted, with access controls). This is essential for both disaster recovery and compliance (ensuring data availability).

By solidifying the database setup and strictly maintaining tenant isolation, you reduce the risk of data leakage between clients and improve overall data security.

Role-Based Access Control (Admin vs. MA vs. Nurse)

The system has a role-based permission scheme (root admin, admin, MA, nurse). We want to ensure **secure interactions between these roles**: - **Root Admin vs. Org Admin:** The root administrator account has unlimited access (manages all organizations), while an org admin manages a single organization. The app enforces separation: if a root admin tries to use an org admin page, it gets redirected to the root dashboard [10]. This prevents a root user from accidentally interacting as an org admin in the wrong context. Continue to ensure that **root admin actions are separated** – e.g., creating an organization or global presets is only via root routes protected by `@root_admin_required` [11]. - **Org Admin Capabilities:** An organization admin can create and manage users in their org, configure Epic credentials, etc. This is appropriate. The code uses `@admin_required` on admin routes which allows both org admins and root admins through (since `is_admin_user()` returns true for both) [12]. This is fine, though you've already blocked root from using those via the redirect logic, meaning effectively only org admins use admin routes. Ensure that only

admin-level users can access sensitive endpoints (the decorators and role checks in place, like `if not current_user.is_admin_user(): abort(403)` [12], accomplish this). - **MA and Nurse Roles:** These are lower-privileged roles. Typically they might only access screening results and patient info, not system settings. Verify that all admin routes (user management, org settings, etc.) require admin role – which they do via `@login_required` + `@admin_required` on the blueprint routes [13] [14]. Non-admin users are redirected to the basic UI dashboard if they attempt admin URLs [15]. This ensures MAs/Nurses cannot load admin pages. Continue to build any new feature for these roles with similar care (use `@require_role('MA')` or similar if needed to restrict certain views to certain roles only [16]). - **Privilege Separation:** There may be cases where an admin should not do something that only a nurse should (though usually admin can do anything a nurse can). In general, admins have superset privileges which is acceptable. Just make sure that **no sensitive information from other roles "bleeds" over** – e.g., an admin viewing logs or data should still only see their org's data. The `AdminLog` queries in the admin dashboard are already scoped to `current_user.org_id` [17], so an admin will only see events from their own org, which is good. - **Monitoring and Alerts:** It's a good practice to log and possibly alert on any abnormal role usage. For example, if a user somehow gains admin rights or a nurse account tries to access an admin endpoint (which would trigger a 403), log that as a security event. The code does log a warning if role access is violated in `require_role` decorator [18]. Reviewing those logs can help detect misuse or attempted privilege escalation.

Overall, the role-based access control in the app is well thought out. Maintaining these boundaries and reviewing the logs will keep the separation between root admin, org admin, and regular users secure.

Username & Password Management

Proper management of user credentials is essential. The application already uses secure password hashing, and we suggest a few additional improvements: - **Password Hashing:** The app uses Werkzeug's password hash functions (PBKDF2 by default) – on user creation it calls `User.set_password()`, which stores a hash of the password [19]. This is correct; plaintext passwords are never stored. Ensure that a strong hashing algorithm (PBKDF2-SHA256 with a salt, as Werkzeug uses) is in place and keep the library up to date for any improvements. - **Strong Password Policy:** Enforce a strong password policy for users, especially admin users. Currently, there's no check on password complexity or length in the code (e.g., an admin could set "123" as a password). Implement requirements such as a minimum length (e.g. 8 characters), and encourage a mix of letters, numbers, symbols. This will reduce the risk of brute-force or guessable passwords. - **Remove Default Credentials:** In the development config, the app inserts a default admin user (`admin/admin123`) if none exists [20]. **Do not use default logins in production.** Before deployment, remove or disable the `initialize_default_data()` that creates these users. A known password like "admin123" is a huge vulnerability if it ever exists in a prod database. If the system has already been deployed with that code, check that no default admin user is present (or it has a securely changed password). Also remove the code that prints those credentials to the console on first run [21]. - **Account Lockout & 2FA:** The app tracks failed logins and locks an account for 30 minutes after 5 failures [22] – good for preventing password guessing. Consider adding **two-factor authentication** for an extra layer, at least for privileged accounts. The `User` model has a `two_factor_enabled` field [23], so this was anticipated. Implementing 2FA (via an authenticator app or SMS/email OTP) would significantly improve security for login, especially for admins. This mitigates risk if a password is compromised. - **Session Management:** The system updates `last_activity` and can enforce session timeouts (default 30 minutes) [24]. Ensure that this is working (the code sets a `session_timeout_minutes` per user). You might also configure Flask-Login's session protection features (like `SECURE_SESSION_REFRESH`) to invalidate sessions after inactivity

or if the user's IP/UA changes. Also, because `SECRET_KEY` signs the session cookies, **make sure SECRET_KEY is a high-entropy value in production** (the default "dev-secret-key..." in code [25] must be overridden via env var). - **User Enumeration:** When logging in, be mindful of error messages – e.g., don't reveal "email not found" vs "wrong password" in a way that attackers can enumerate valid usernames. Ensure the UI messages are generic like "Invalid credentials" for both cases.

By tightening password policies, eliminating any default credentials, and possibly introducing 2FA, the system's authentication will be much more resilient to attacks.

Secrets and Credential Management

The application handles various **secrets and credentials** – e.g., app secrets, Epic API credentials, JWT keys, etc. We need to manage these carefully: - **App Secret Key:** The Flask `SECRET_KEY` is used for sessions and CSRF. In production this must be set to a long, random value via an environment variable (and rotated if there's any suspicion of leak). The code currently falls back to a hardcoded `"dev-secret-key-change-in-production"` [25]; double-check that in your deployment an env var is overriding this. Never commit real secret keys to the repo. - **Epic Client Secret Storage:** Each Organization has an `epic_client_secret` field in the DB [26]. Right now, when an admin enters it on the Epic registration page, it's stored in plaintext (the code just assigns it directly) [27]. **This is risky** – if the database were compromised, the OAuth client secret for Epic could be used. Ideally, perform application-level encryption for this field: for example, encrypt it using a symmetric key stored in an environment variable or key management service. Then store the ciphertext in the DB (and decrypt on the fly when needed for OAuth). At minimum, add a note that in production the `epic_client_secret` is encrypted – the code even comments "should be encrypted" [28]. Now is the time to implement that encryption. This also applies to the `EpicCredentials` tokens: access and refresh tokens should be encrypted at rest rather than plain text [29]. - **Remove Hardcoded Keys:** We found an RSA private key checked into the repo (`np-key-2025.pem`). This is presumably for JWT client assertions to Epic. **Do not store private keys in the repository.** Instead, load them from secure environment variables or a secrets manager. For example, the app's `JWTClientAuthService` is designed to fetch keys from env vars (looking for `NP_KEY_...` prefixes) [30], which is good. You should use that and not rely on a file in code. The presence of `-----BEGIN PRIVATE KEY-----` in the repo is a red flag [31]. Rotate this key (generate a new key pair) since it may be compromised, and use the new key via environment configuration. - **Key Rotation:** Implement a schedule and mechanism for rotating sensitive keys. For instance, the JWT signing key for Epic – you might generate a new RSA key annually (the naming `np-2025-08-a` suggests a date/version). The system supports multiple keys (via the JWKS endpoint and `kid`), so you can load a new key alongside the old and update the `kid`. Plan a rotation policy for the Flask `SECRET_KEY` as well (rotating that would invalidate sessions, so do it during maintenance windows if ever). Database credentials and Epic client secrets should also be rotated periodically (and whenever a team member with access leaves, etc.). - **Limit Exposure of Secrets:** Ensure that secrets are never exposed in logs or error messages. For example, if an Epic token refresh fails, do not log the token or client secret in plaintext. The code appears to log only high-level errors (which is good). Also, when displaying configurations in the UI, never show the actual Epic client secret to users after it's been saved (the current UI likely doesn't – typically it would show it as blank or masked). - **Use Vault or Env for Config:** Continue to use environment variables for any config secrets (DB passwords, API keys, email SMTP credentials, etc.). For deployment, use Replit's secret manager or a `.env` file which is not committed. The project structure suggests a `.env` file for local secrets [32] – ensure this is used properly and not checked in.

By instituting strong encryption and secret management practices, you reduce the risk that a breach of the database or code repository will expose sensitive credentials. This protects both your application (from unauthorized access to third-party APIs) and patient data.

Code and Database Redundancies Cleanup

As part of security hardening, it's wise to **remove any unnecessary or redundant code and data** that could introduce risk: - **Eliminate Obsolete Files:** We noticed `admin_routes_backup.py` in the repository – a legacy copy of admin routes [33]. Such files can cause confusion and potentially be loaded accidentally. Remove outdated scripts/backups from the production codebase. This reduces attack surface and maintenance burden. - **Secure Database Initialization Scripts:** The project has dev scripts like `reset_db.py` [34], `migrate_multi_tenancy.py`, and test data setup. Ensure these are not executed on a live environment. For instance, `reset_db.py` drops all tables – accidentally running that in production would be catastrophic. Limit access to these scripts or remove them entirely from the deployed package. If you keep a maintenance script, add safeguards (like environment checks or an extra confirmation step) to prevent misuse. - **Remove Test Data:** If any test or demo data remains in the database (e.g., patients or documents used during development), purge it before going live. This avoids accidental exposure of fictitious or irrelevant data and ensures you start with a clean, minimal dataset. - **Default Admin/User Cleanup:** As mentioned, remove the default `admin@example.com` user creation logic [20] from production. Also check for any similar shortcuts (the code also made a default basic user "user/user123" for example [35]). These should not exist in a real deployment. - **Console Logging:** Scan the code for any `print()` or debug logs that might dump sensitive information. For example, after creating default users, the code prints their passwords to the console [21]. Such outputs should be removed to avoid leaving secrets in any log files. Use structured logging (via Python logging library) and log at appropriate levels (info/warning/error), avoiding verbose debug prints in production. - **Optimize Resource Usage:** Redundant code can also lead to redundant processing. For instance, ensure there aren't duplicate operations (like two places where similar queries run). Removing unused code paths (the backup routes, old functions) will make the app leaner and less error-prone, which indirectly improves security (fewer chances for an unmaintained piece of code to become a vulnerability).

By cleaning up the codebase and database, you not only improve security but also maintainability. The system will be easier to audit when only relevant code and data are present.

Epic OAuth Integration (Organization Credentials)

Each organization in the system can connect to an Epic EHR via OAuth2 (SMART on FHIR). We need to ensure this integration is done securely and correctly: - **Required Fields:** As suspected, the key pieces of info for an Epic OAuth setup are the **Client ID**, **Client Secret**, and the FHIR API base URL for that org's Epic instance. The `Organization` model has fields for these [26], and the Epic registration form expects exactly those inputs. There are also fields for `epic_oauth_url` and `epic_token_url` (and endpoint IDs) for production use [36]. Make sure that for each organization, these are populated with the values provided by that client's Epic setup. The sandbox defaults are in the code (for example, `https://fhir.epic.com/interconnect-fhir-oauth` for sandbox) [37], but for production each org will have unique endpoints – ensure the UI allows entering those (the root admin edit page might be used for that [38]). - **Secure Storage of Credentials:** As noted in *Secrets Management*, encrypt the `epic_client_secret`. Also consider whether organization admins should be able to see the secret after it's saved – typically, you would **not**

display it back in plaintext. If the UI currently does (for example, in the root admin org edit page, it might show the field), change it to blank out or mask the secret. Treat it like a password field. - **OAuth Redirect URI:** Ensure that the redirect URI used in the OAuth flow is correct and uses HTTPS. The code constructs the redirect URI dynamically and even forces it to https:// if it mistakenly was http:// [39] [40]. This is good. Verify that the redirect URI configured in Epic App Orchard exactly matches the one your app uses (including path and case). Any mismatch will cause authorization to fail – this is more of a deployment detail, but worth noting. - **Least Privilege Scopes:** The scopes requested for Epic (as seen in the code, they include patient/Observation.read, etc.) should be limited to only what's needed [41]. This looks appropriate for reading patient data and writing the prep sheet DocumentReference. Avoid requesting unnecessary scopes. This minimizes the impact if an OAuth token is ever misused. - **Token Handling:** The app stores the OAuth tokens in the `EpicCredentials` table [29] for reuse by all users in the org. That's convenient but make sure to protect those tokens (encrypt them at rest, and perhaps expire them if an org disconnects). The code sets `org.is_epic_connected=True` when a token is stored and provides a disconnect route. Consider adding a **"Disconnect"** action for org admins that not only clears the session tokens (which you do [42]) but also deletes the stored tokens from the DB so that integration is fully reset. - **Epic Credentials per Role:** Only org admins (and root) should manage Epic credentials. The routes in `oauth_routes` are protected by `@require_admin` [43], so nurses/MAs can't initiate OAuth flows – good. Maintain that: the people handling client IDs and secrets should be limited. - **Logging and Monitoring:** Log when an Epic connection is established or fails. The code logs successes and failures for OAuth [44]. Also, the `EpicSessionCleanupService` and test endpoints are in place to handle issues like multiple sessions [45]. Continue to monitor those logs to troubleshoot any authorization issues quickly – often misconfiguration of credentials is the cause.

In summary, **client ID, client secret, and FHIR base URL** are the necessary inputs for Epic OAuth (plus the redirect URI which is fixed by the app) [27]. Treat the client secret with care, and ensure each organization's credentials are kept separate and secure. This will allow the Epic integration to remain robust and compliant (Epic's APIs themselves are HTTPS and require their own agreements, but that's on the hospital's side).

Replit Deployment Considerations

Since the application is currently deployed on Replit, we should address resource management and deployment config: - **Database Persistence:** Replit provides a PostgreSQL add-on (as indicated by the Nix config). Ensure that the Postgres instance is properly persisted. If using the local filesystem for the database (or worse, SQLite), data might not persist through reboots or scaling. Verify that the database lives on a persistent volume or an external service. For production, you might even use an external managed database (which can offer better performance, backups, and HIPAA compliance if through a provider like AWS RDS with a BAA). - **File Storage:** If the app stores uploaded files (e.g. PDFs of medical documents), Replit's file system might not be ideal for large or numerous files. It's constrained in space and may not survive instance resets in autoscale mode. Consider using a cloud storage service (Amazon S3 or similar) for file uploads, with objects encrypted at rest. At minimum, monitor your Replit disk usage so it doesn't fill up with uploaded documents or logs. Remove any temp files promptly. The `uploads` folder should be on persistent storage if you expect files to persist. - **Memory/CPU Limits:** Replit imposes limits on CPU and memory depending on the plan. A screening engine could be heavy (OCR with Tesseract, etc.). Be mindful of these limits – for example, avoid loading very large data sets into memory. Use streaming or pagination for large queries. The background tasks (if any) might need tuning to not overwhelm the environment. - **Autoscaling & Multi-Instance:** If Replit Autoscale spawns multiple instances, ensure your design can handle it. For example, with multiple app instances, using a local SQLite would break (since each instance

would have its own copy). Using Postgres mitigates that, but then consider a distributed cache for any in-memory data if needed. Also, session management might need to be server-side or use a shared Redis if you scale out (Flask's default cookie sessions will work as long as SECRET_KEY is same on all instances). - **Backups and Recovery:** Replit doesn't automatically provide database backups. Implement your own backup routine. This could be a cron job (perhaps using the Replit "Always On" scheduler or an external service) that dumps the Postgres DB daily to a secure off-site location. Also, download periodic snapshots of any uploaded files. This ensures you can recover in case of accidental data loss. - **Transition Plan:** Given Replit is not suitable for real PHI, plan the transition to a compliant environment sooner rather than later. If you must demonstrate the app on Replit, use only dummy data. Meanwhile, set up an AWS or other cloud account for the production deployment. Many of the improvements above (encryption, env vars, etc.) will carry over. On AWS, for example, you can use services like AWS Secrets Manager for client secrets, RDS for Postgres (with storage encryption), and S3 for files, all under a BAA. This will satisfy HIPAA's requirements more fully than Replit can.

Proper resource management and deployment practices will ensure that the app runs reliably and securely. While Replit is fantastic for development and quick demos, a healthcare application with compliance requirements will eventually need a more enterprise-grade deployment. Keep the development environment and production environment configurations separate, and apply all the security best practices in both, with the understanding that **production must meet higher standards (HIPAA/BAA, redundancy, monitoring)** than a dev sandbox.

Conclusion

By implementing the above recommendations, the HealthPrep v2 system will significantly improve in security and compliance:

- Enforce HTTPS and secure cookie practices to protect data in transit.
- Move to a HIPAA-compliant hosting infrastructure (since Replit won't sign a BAA) [46] and use encryption and auditing to satisfy HIPAA safeguards.
- Strengthen database usage with proper multi-tenant isolation, backups, and by removing any default or test data.
- Maintain strict role-based access control, ensuring each user role only accesses authorized data.
- Improve authentication security with strong passwords, removal of default creds, and possibly two-factor auth for admins.
- Manage all secrets (app keys, Epic credentials, JWT keys) outside of code, encrypt them at rest, and rotate them regularly to minimize exposure.
- Clean up the codebase and scripts to remove any unnecessary risk and make the application lean and easier to secure.
- Handle the Epic integration credentials with care, and verify each org's configuration while protecting the OAuth secrets.
- Finally, optimize the deployment setup so that the app's resources are managed properly and the environment meets the necessary compliance standards.

With these changes, the application will be far more robust against breaches and aligned with healthcare industry security practices, giving confidence to proceed with handling real patient data securely. [3] [19]

---

1 2 20 21 35 37 settings.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/config/settings.py

3 46 Is Replit.com HIPAA compliant? (2025 update)
https://www.paubox.com/blog/is-replit.com-hipaa-compliant-2025-update

4 5 7 19 22 23 24 26 36 models.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/models.py

6 10 15 25 app.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/app.py

8 16 18 multi_tenancy.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/utils/multi_tenancy.py

9 11 28 38 root_admin_routes.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/routes/root_admin_routes.py

12 13 14 17 admin_routes.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/routes/admin_routes.py

27 41 epic_registration_routes.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/routes/epic_registration_routes.py

29 39 40 42 43 44 45 oauth_routes.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/routes/oauth_routes.py

30 jwt_client_auth.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/services/jwt_client_auth.py

31 np-key-2025.pem
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/np-key-2025.pem

32 folder_structure.txt
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/attached_assets/folder_structure.txt

33 admin_routes_backup.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/routes/admin_routes_backup.py

34 reset_db.py
https://github.com/mitfusco98/HealthPrepV2/blob/ecd308aa61c685848a68ae5060ad34ca858be31c/reset_db.py